DPMax – A Tool for Textual and Graphical Analyses of Common Dynamic Programming Algorithms.

Part 2 – The *Longest and Shortest Paths Problems using Weights*

CHRISTIAN COLOSSUS

GRADUATE DIPLOMA THESIS

BIOINFORMATICS FOR SOFTWARE PROFESSIONALS PROGRAM

CENTENNIAL COLLEGE, TORONTO

SECOND EDITION

First Edition Spring 2009

CONTENTS

ABSTRACT

The first part of this paper detailed the workings of the LCS module of DPMax version 1.0. DPMax is a software tool that was written to implement and display the ways common dynamic programming (DP) algorithms work. These DP algorithms not only returned the text output of the execution of their algorithms, but also displayed in a graphical format the resulting matrixes after the execution of those algorithms. DPMax contains two fully finished modules – Weights and Longest Common Subsequence.

The version 1.0 of DPMax for Weights implements the DP algorithm that solves the problem of the longest paths between 2 points given a series of values. These values represent the distances between adjacent cities or geographical locations. It is modelled closely on the Longest and Shortest Paths Problem as in the *Traveling Salesman* and *Manhattan Tourist Problems* which are detailed in many Mathematical and Computer Science texts (Reingold et. al., Dasgupta et. al.).

DPMax for Weights makes the function of achieving the longest and shortest paths easy to comprehend and understand, just like its LCS counterpart, with its text and graphic outputs. The text output with weights simply displays the pairs of rows and columns which represent the edges and vertices of the graph used to solve the problem and then display the graphical solution.

INTRODUCTION

So much has been said about DP in the first part of the paper in the writeup on the LCS module. So, this part of the paper will go straight to the point.

Dynamic programming is a mathematical-cum-computer scientific method to solve certain kinds of optimization problems. These unique problems must have certain properties (Bellman) including:

1. It should obey the principle of optimality which means that
a. It should comprise overlapping subproblems
b. It should exhibit an overlapping substructure

2. It should be visualizable as a matrix comprising smaller problems with each one having the same or very similar properties, like the decision variables to be solved, as the main problem. In other words, each of the constituent problems should be a mini-version, or mini-repetition, of the big one.

As soon as the dynamic programming is confirmed, an appropriate DP algorithm must be sought to solve the problem.

The two methods used for the implementation of DP are the classic method (aka top-down or memoization method) and the bottom-up method (sometimes called the tabulation method). DPMax employs the bottom-up method for its modules.

It should be noted that there will be no repetitions from the LCS paper. So, comments shall be made in this paper only when there is a marked difference in a particular section of the two modules.

Solving the Longest Paths Problem using Graph Weights (The Manhattan Tourist Problem)

The problem of the Manhattan Tourist lies in making a decision – how to traverse the many possible paths between the 2 extreme and opposing points while seeing as many tourist attractions as possible. From the northwesternmost point (called the ***source*** vertex) the traveller must eventually get to the southeasternmost point (called the ***sink*** vertex). The distance from one vertex to another is called a weight. The weight of a path from the source vertex to the sink vertex is calculated as the sum of weights of its edges. These weights are the total number of attractions which the tourist desires to see.

The intersections between the streets that run from west-east and north-south form the vertices while the edges are the streets running north-south and west-east which intersect with one another. This structure of edges and vertices is a grid-like graph, *G*. The horizontal edges in G run from west to east and the vertical one run from north to south. All the edges have weights which are the length of the streets as they run from one vertex, or intersection, to another. A continuous sequence of edges is called a path. The length of each path is the sum of all the weights of the edges in that path.

What is the definition of the Manhattan Tourist Problem?

To find the greatest (maximum) number of attractions between the source and sink vertices. This path is the longest path, and the largest weight, between the 2 vertices (i.e. the start and finish points) (Dasgupta et. al.).

Input: A graph with weighted edges from west to east and north to south.

Output: The longest path with largest weight, *L*, through the grid, *G*, from the source to sink vertices.

Solution

1. The total number of paths running from west to east represent the number of rows in the grid while the number of paths running from north to south are the columns in the grid in the graph.

Let the number of rows = n, while

the number of columns = m,

We are to find $L_{n, m}$, the length of the longest path with the maximum weight through *G*

2. The source vertex is at position *G*[0, 0] of the grid while the sink vertex is at *G*[n, m].

3. The total number of subproblems is (*n x m*) which represents the total number of all vertices in *G*. We must start by finding all the weights of vertices along the first row, *G*[0, j], and first column, *G*[i, 0]. Because the tourists can only move eastwards from the source to sink, then *G*[0, j] (for 0 <= j <= m) is the sum of weights of the first j city blocks. Likewise, *G*[i, 0] (for 0 <= i <= n) is the sum of weights of the first i city blocks, because the tourists can only move southwards from source towards sink.

4. Now that we have found the base case of the Weights algorithm, we must now proceed to find the lengths of the paths between the vertices along the rows and columns. This can be solved by a recurrence that takes into consideration the edges and vertices and the movements between them. We know that getting to any vertex in the grid can only occur by moving southwards, (from *G*[i, j] to *G*[i-1, j]), or eastwards (from *G*[i, j] to *G*[i, j-1]), through the grid. We can formulate the solution through the subproblems of the grid as:

$L_{i, j}$ = max ($L_{i,-1, j}$ + weight of the edge between *G*[i -1, j] and *G*[i, j],

$L_{i, j-1}$ + weight of the edge between *G*[i , j-1] and *G*[i, j],)

5. With the foregoing, we can formulate the Longest Path by Weights (Manhattan Tourist) algorithm

function LongestPathByWeights ()

H[] – A 2-Dimensional array of weights of edges that run from west to east (horizontal)

V[] – A 2-Dimensional array of weights of edges that run from north to south (vertical)

n = length of the array *H[]*

m = length of the array *V*[]

(1) *G*[0, 0] = 0

(2) for i = 0 to n

G[i, 0] = *G*[i -1, 0] + *V*[i, 0]

(3) for j = 0 to n

G[0, j] = *G*[j -1, 0] + *H*[0, j]

(4) for i = 1 to n

(5) for j = 1 to m

(6) $L_{i, j}$ = max ($L_{i,-1, j}$ + *V*[i, j], $L_{i, j-1}$ + *H*[i, j])

(7) return $L_{n, m}$

6. Again, as in the LCS module, careful attention must be taken so as not to do a simple max() function to obtain the larger path between two possible paths. If two paths have equal lengths, both paths must be marked as possible paths to the optimal solutions. This may mean that we have multiple optimal paths to the sink vertex, that is, the ending point for the tourist.

7. Now that we have the optimal path or paths, we have to develop a backtracking algorithm to obtain those paths from source to sink.

Computational Complexity

We have already established that we have a fixed number of subproblems obtained as the product of the rows and columns, that is, *n* x *m*. the beginning of the Weights function takes as input 2 2-Dimensional arrays, *H* and *V*. Both are arrays of arrays of the weights of the Horizontal (west-east) and Vertical (north-south) edges, respectively. From lines (1) to (3) are the initialization conditions of the algorithm. Lines (4) and (5) are the outer and inner loops over the horizontal and vertical arrays, respectively. The runtime of the first loop, over the horizontal array of west-east weights, is O(*n*), while the runtime of the second loop over the vertical array of north-south weights is O(*m*).

The basic operation within the inner loop is a max() function which costs a simple constant O(*1*) time. Even if there is a check for equal paths, the operation cost doesn't change, it is still constant. So, the total runtime through the ManhattanTourist function is O(*nm*) x O(*1*), which comes to O(*nm*). If both rows and columns have the same length such that *n* = *m*, then the runtime is O(n^2).

The backtracking function should be as simple as possible because the complete backtracking operation costs as much as exponential. But this implementation is a modified backtracking operation that obtains only 1 optimal solution path to avoid the usual complexities with this operation. It starts from the sink vertex and follows the path of its subproblems of solutions until it gets to the source vertex and then it exits, returning the optimal longest path from source to sink vertices. As in the LCS, each vertex at *G*[i, j], has a store of which prior vertex provided its solution. There are no diagonal vertices here unlike in the LCS, so there can only be a path from the north vertex, *G*[i-1, j], or the west vertex, *G*[i, j-1]. The modified algorithm runs in O(*k*) where *k* is the number of vertices (intersections) traversed to obtain the longest path $L_{n, m}$.

METHODS AND EXPERIMENTS

DPMax for Weights was used to compute the modified Manhattan Tourist Problem by finding the longest paths between 2 points – a starting point and a finish point.

The grid was a 4-by-4 grid with 4 rows and 4 columns. The number of rows and columns were input in the Input tab and the 'OK' button was pressed. The GUI produced a dialog input box that prompted for each of the west-east edge weights one by one, specifying the particular intersection of rows (row i and row i+1) that should be entered. At soon as the input box captured all the west-east weights data, it prompted for the same weights for the north-south edges one by one.

Once that was completed, the software performed the computation of the Weights algorithm as detailed earlier. The Output tab would then have the results of the analysis in text format.

Also, the Graphical tab contained the graph matrix that was produced by the 4x4 grid specified in the input tab. The grid comprises circles connected by arrows or lines and then numbers along the arrows or lines or in the circles.

The solution to the grid is displayed in 3 parts –

1. The lines or arrows that connect the circles are the weights of edges that represent the streets of the city,

2. The circles represent the vertices between the edges. They are the intersections between the streets. The first circle at the northwesternmost end of the graph is the source vertex, and it is the starting point for the tourist, and,

3. The numbers by the edges represent the weights of the edges. The numbers in the circles are the solutions to the vertex subproblems. They are the sum of the longest path leading to that particular vertex.

3. The lines connection the vertices may be either one of 2 forms

a.) An arrow pointing from one vertex to another indicating that the vertex the arrow is pointing to is a solution to the vertex it is pointing to.

b.) A line connecting 2 vertices indicating that it's not a solution to the connecting vertex.

There could be multiple arrows pointing to a vertex meaning that both connecting vertices have produced solutions to that vertex. In a simple grid, a visual backtracking operation could be performed to find the longest paths from the source to the sink.

The arrows and numbers shown in each cell were produced via repeated calls to the Java Graphics methods draw() and others to render the images on a JPanel canvas as follows:

1. The grid is drawn as a 2-dimensional array of arrays of edges running from west-east and north-south, and vertices intersecting the edges.
2. The vertices of the graph are drawn as circles with each connected to surrounding vertices in the matrix by edges.
3. The edges were drawn as line connecting the vertices. the west-east weights are drawn first, then the north-south weights are drawn.
4. The solutions of the vertices were computed and arrows were drawn to signify what vertices are the solutions to others, if any exist.
5. The numbers are drawn beside or above the edges to show the weights of edges and also inside the circular vertices to indicate the solution for each vertex.

When completed, a grid of possibilities appears for the tourist to traverse the longest path of maximum interactions, which is the optimal solution to the problem (Dasgupta *et al*., Chap.6).

RESULTS

DPMax for Weights creates an *n* x *m* matrix, a grid of intersecting edges and vertices, comprising a graph *G*[*n, m*]. The input into the GUI was a set of 2-Dimensional array of weights – the first was the array of arrays of west-east weights while the second one was an array of arrays of north-south weights. The final set of weights which were input by the user were displayed on the respective text areas in the Input tab of the Weights module.

The length of the west-east 2-D array is *n*, while the length of the north-south array is *m*. All the input values must be integers because that is the expected input. The first vertex, the source, has the value of 0. This is because it is the starting point and has no incoming edges. The sink vertex contains the longest path as computed by the algorithm.

There were two sets of experimental grids created with DPMax – one was a 4x4 grid while the other one was a 5x5 matrix. For the sake of brevity, the results of the 4x4 matrix is presented here. The results of the other, larger, grid is in the Appendices. It should be noted that the software can create much larger grids.

The Results for 4x4 Weights Grid

The result of a 4x4 matrix of edge weights and vertices was determined. The output of the results was displayed in text format on the 'Output' tab and the graphical display was presented on the 'Graphics' tab.

The Input tab is shown in the figure below, and it shows the number of rows and columns specified in the appropriate input box in the Weights GUI.

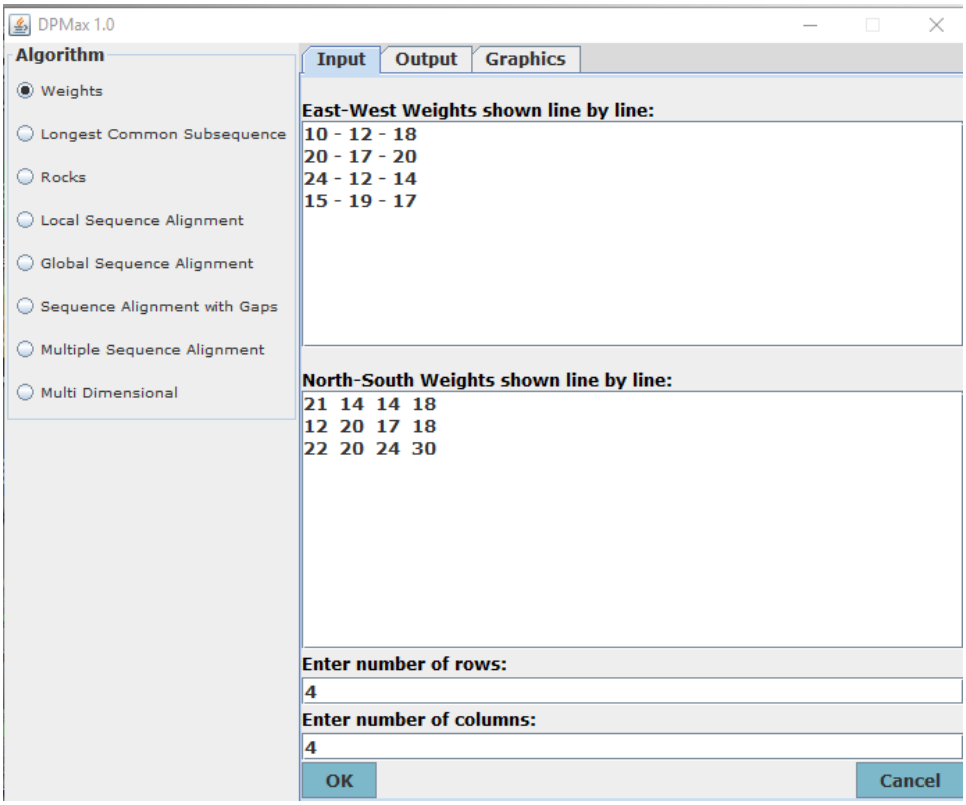


Figure 1: The Input tab of the Weights module. It shows the input number of rows and columns. The values of the west-east and north-south weights input by the user are displayed in the upper and lower text areas, respectively.

The next figure shows the Java Input dialog box which accepts the input values for the weights of both west-east and north-south edges.

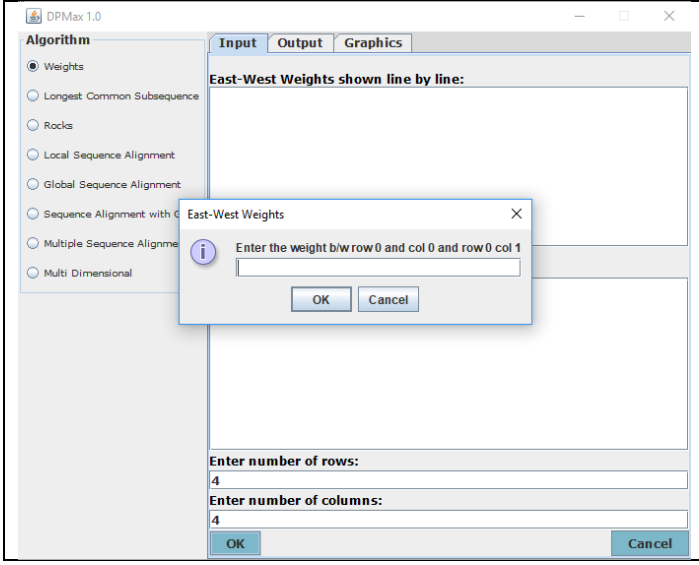


Figure 2A: The Java Input dialog box for accepting user input for the West-East weights of edges.

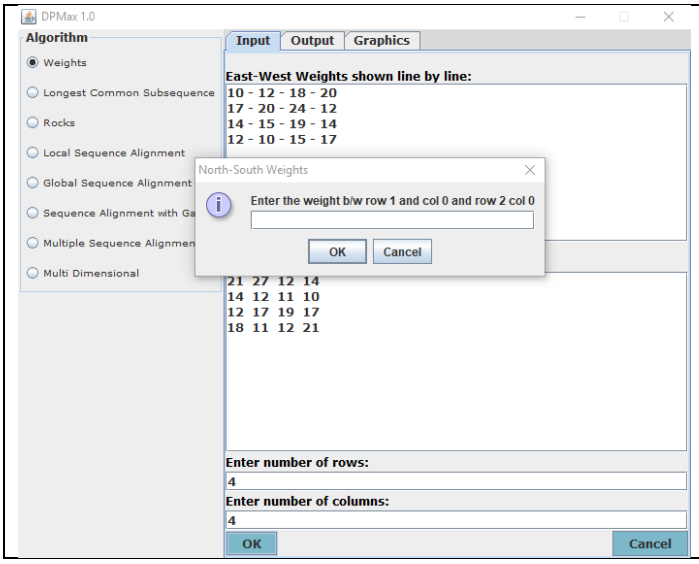


Figure 2B: The Java Input dialog box for accepting user input for the North-South weights of edges.

Once the user input is complete, the Output tab will contain the results of the algorithm. This result would be the longest path between the source and sink vertices in text in the format, [0, 0]…[i, j]…[n, m]. The longest path from the start to end of the 4x4 grid matrix is shown in the next figure that shows the results of the Output tab of the Weights module.

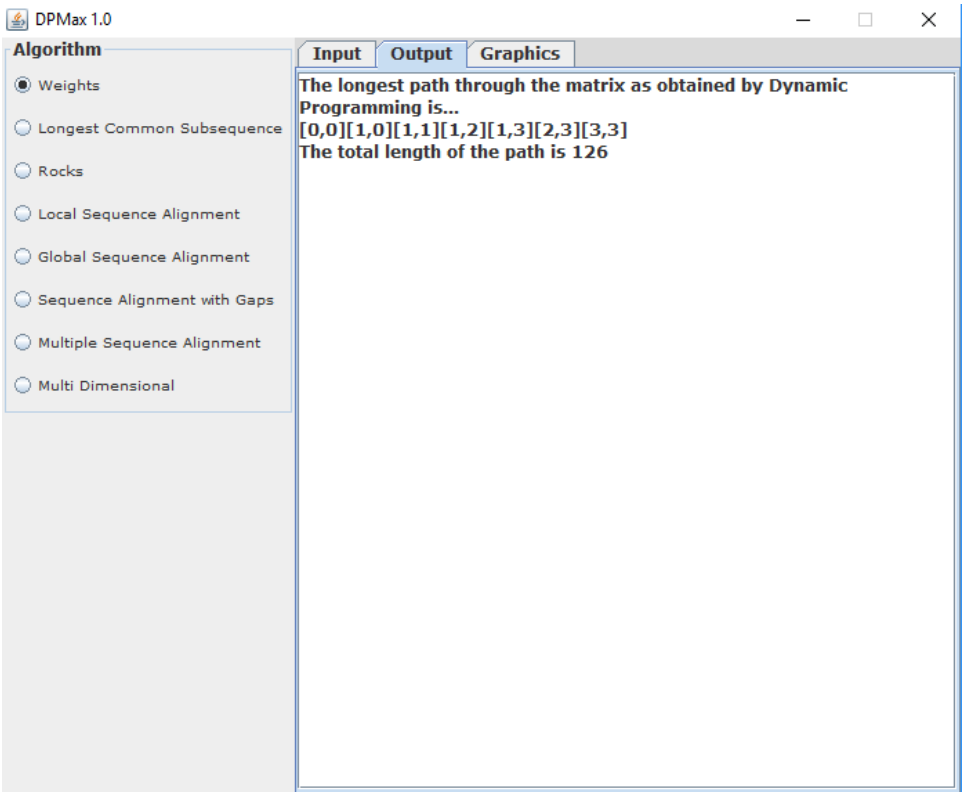


Figure 3: The Output tab of the Weights module. It shows the longest path through the matrix of the 4x4 grid. The longest path through the grid, $L_{n,m}$, is 126.

Graphics Output

The figure below is the graphical display of the 4x4 grid produced by DPMax.

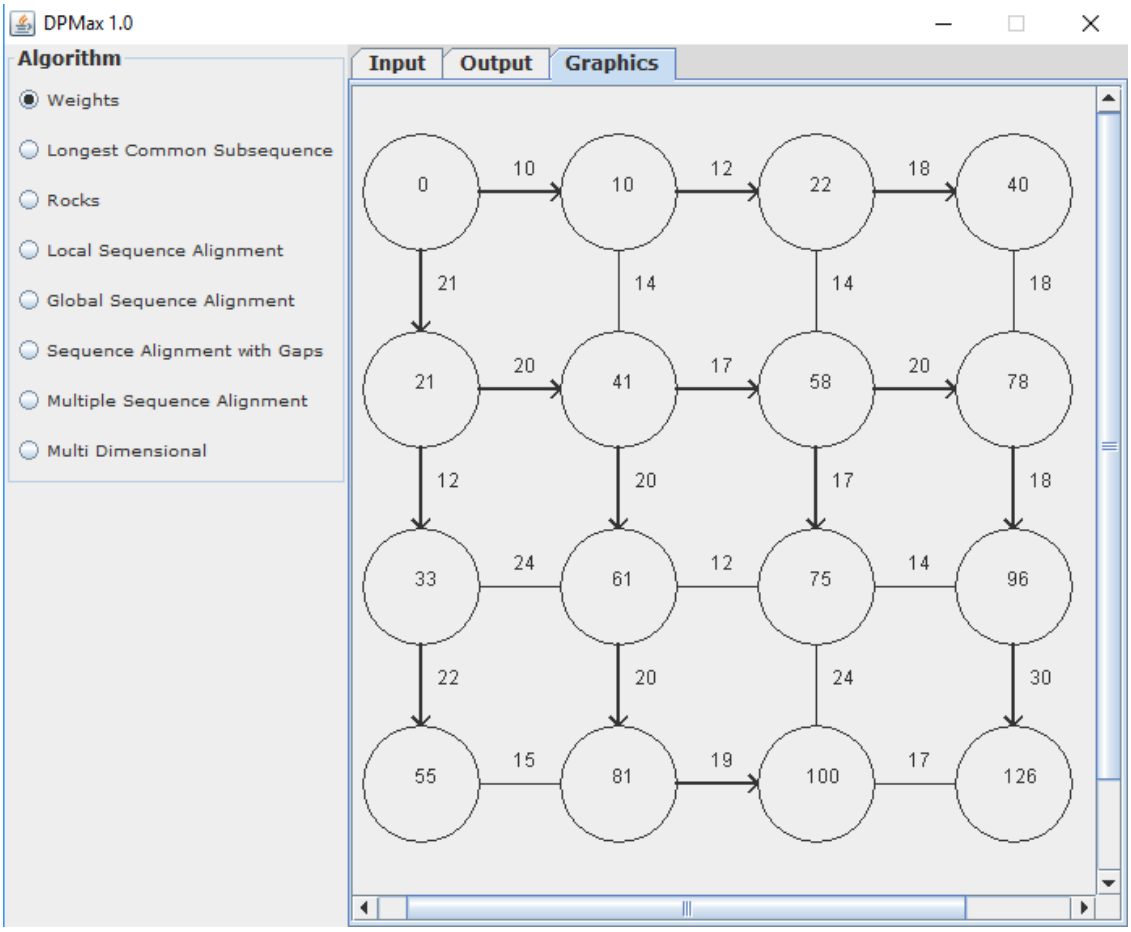


Figure 4: The Graphics tab of the Weights module. The matrix displays a 4-rows-x-4-columns grid with the optimal solution at the position *G*[n, m]. The longest path ([0,0][1,0][1,1][1,2][1,3][2,3][3,3]) has the value of 126.

The longest path is ([0,0][1,0][1,1][1,2][1,3][2,3][3,3]) has a value of 126. The source vertex the first one and has a value of 0. The last vertex, the sink, has a value of 126. The vertex array of the longest path is ([0][21][41][58][78][96][126]).

Architecture, High-level Design and Program Flow

The architecture, design and program flow for the Weights module are the same as the LCS module. The Object classes, however, are different and they are as follows:

1. DPMaxController makes a call to DynProgWeights class, which has the code to create the GUI for the Weights module and also contains the functionality that handles the Weights algorithm itself.

2. The panel on which the grid is drawn is the WeightsDrawPanel class.

On the Weights panel, there are 3 possible GUI interactions

A) Input: For the input of the grid to be created.

B) Output: For the display of the result of the longest path through the matrix in text format, and,

C) Graphics panel: This is the display of the result of the grid in graphic format which displays the matrix.

Clicking on the button ‘OK’ on the input page calls the actionPerformed() method of the Weights View which hands control to the class DynProgWeights class.

The *DPMax for Weights* Module

The Weights Module Methods and Functions

The main class that contains the Weights module view and functionality is the DynProgWeights class. The ***dpg*** object is an instance of, and encapsulates the functionality within, the DPGraphics class.

The Input tab has an 'OK' button which when pressed triggers an action event that is sent to the actionPerformed() method in this class. Once in this method, the class reads the 2 numbers for rows and columns. The class then builds a grid according to the specified number of rows and columns. It uses these numbers to create an input dialog box which prompts the user to enter the west-east weights according to the rows and columns needed to build the grid.

Once it does that, it calls the method doDPWeights() which reads the number of rows and columns and creates the grid matrix based on those numbers. The grid is just a list of objects of type Cell as encapsulated within that class. This means that each vertex in the matrix is of type Cell and these cells are connected by lines or arrows.

Initializing the matrix scores

The doDPWeights() method then calls the method initialize() to initialize the matrix and it does so by creating each vertex cell (the first of which is created with an initial score of 0) and positioning them in the matrix. The initialize() method promptly calls helper method initializeScores() which will set the scores for each cell in that grid for the first row, west-east weights, and first column, the north-south weights. The initialize() method will first present the user with the option input box that will accept the values of the edge weights from the user. Once that is done, it parses them as integers and places them in the appropriate 2-Dimensional arrays – one for the west-east and another for the north-south. At this point, initialize() calls initializeScores() which will compute the scores for the vertices in first row, $G[0, j]$, and the first column, $G[i, 0]$. Then all the other vertices in the grid's matrix must be filled.

Filling the LCS Matrix: fillMatrix()

Once the initial scores have been fixed in each cell in the matrix, doDPWeights() then calls another method fillMatrix() to actually fill the matrix with the solution of each cell until it reaches the optimal solution at the sink vertex. And because this is the bottom-up approach, just like the LCS module, it starts by solving the cells which are the smaller problems and then uses the solutions for those cells that have been solved to find the solution for the overlapping cells which are larger problems. But unlike the LCS algorithm, it progresses downwards

towards the south and eastwards until it solves the sink cell which is the final vertex/intersection.

This method, implements the routines of the Weights recurrence algorithm. It begins by going into a set of ***for*** loops thereby incurring the bulk of the runtime cost. The first loop iterates over the array of west-east weights. The second, inner loop, iterates over the array of north-south weights. Within this code, the O(*nm*) runtime of the algorithm occurs. The basic operation of solving each subproblem is done by the helper method that is called within this fillMatrix() method. That method, fillCell(), performs basic operations in constant O(*1*) time.

By the end of fillMatrix(), each vertex of the matrix has been given a solution score and the final vertex has been scored. fillMatrix() delegates that functionality to another method fillCell() to do the actual cell filling. fillCell() is a helper method which fills cells of the matrix by the Weights algorithm being used. After being solved, every cell is then connected to the surrounding cells – above, below, left, or right. Every cell must know its relative cells in the optimal substructure of *G*.

Filling each cell: The fillCell() method

The function of the fillCell() method is quite straightforward. It is an overloaded method which takes as parameter 3 Weights cells. These cells are the current cell, the cell above and the cell to the left of the current.

With these parameters, it builds connections and pointers between the cells as needed. It connects each cell to its surrounding cell and creates pointers that indicate which cell or cells provided the solution to the current cell. Every cell has pointers set to indicate where its solution came from and every cell also contains a class member which is an instance of java.util.Map where it stores all its connections to its surrounding cells.

Once the matrix has been filled, we know its optimal solution in the sink vertex. But the task is far from completed. We still need to find the actual path that ends in the optimal solution at the sink. This is the final step in many dynamic programming programs and for the Longest Path by Weights Problem, it is what gives us the actual path that we have been searching for – the longest path it will take for the Manhattan Tourist to experience as many attractions as possible. All this is done through something called a *traceback* – a backtracking technique for finding the final solution of our Weights matrix that contains the optimal solution so found. It is aptly called a traceback because it does exactly what its name implies – it walks backwards through the matrix starting at the optimal solution at the sink, to the beginning of the matrix at the source.

The Backtrack function: doTraceback()

This code for the doTraceback() function is quite the same as the LCS. The only differences are: firstly, here the function returns the longest path from source to sink as a String object which contains a series of vertices expressed as intersections of rows and columns ([row, column]). Also, the final vertex, the source, is inserted at the front of the String once the traceback reaches the beginning of the grid. Once the traceback is complete, then the longest path from source to sink would have been achieved.

This traceback is a simplified version of a typical traceback. It starts at the sink vertex and regresses through the list of previous cells until it gets to the end of the list where if finds a null cell or a cell with a score less than or equal to 0. Once there is a null or zero/subzero score, it has reached the end of this list, or at the source vertex and it exits the while loop. One it does that, it inserts the source vertex at the start of the list. The traceback queries every cell to get the previous cell, that is, the cell which provided the solution to the current cell. Once it gets that information of that current cell, it sets the previous cell as the current cell and the process repeats itself. It keeps on looping until if finds a current cell which has the previous cell as null or cell score not greater than 0. By this time, it has reached the source vertex. The previous cell could be the cell to the left (west) or the cell above (north).

```
public static void initialize() {
        //get the weights that we need for the whole table
        String sWeight = "";
        Cell prevCell = null;
        Cell currCell = null;
        Map<String, Cell> prevCellMap = new HashMap<String, Cell>();
         for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++)
            {
              matrix[row][col] = new Cell(row, col);
              if(row == 0 && col > 0){
                  currCell = matrix[row][col];
                  prevCell = matrix[row][col-1];
                  currCell.setPrevCell(prevCell);
                  prevCellMap.put(Constant.CELL_TO_LEFT, prevCell);
                  currCell.setPrevCellMap(prevCellMap);
              }

              if(col == 0 && row > 0){
                  currCell = matrix[row][col];
                  prevCell = matrix[row-1][col];
                  currCell.setPrevCell(prevCell);
                  prevCellMap.put(Constant.CELL_ABOVE, prevCell);
                  currCell.setPrevCellMap(prevCellMap);
              }

            }
        }

        rows = "";
         //fill up the west-east weights
          for (int row = 0; row < row_weights.length; row++) {
            for (int col = 0; col < row_weights[row].length; col++) {
                  try {
                              sWeight = JOptionPane.showInputDialog(null,
                                          "Enter the weight b/w row " + row
                                          + " and col " + col + " and row "
                                          + row + " col " + (col+1), "West-East Weights",

                              JOptionPane.INFORMATION_MESSAGE);
                              } catch (HeadlessException e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                                    return;
```

```
                }


                try {
                        row_weights[row][col] = Integer.parseInt(sWeight);
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                        return;
                }

                 if(col < nCols - 2)
                        rows += sWeight + " - ";
                 else
                        rows += sWeight;
        }
  rows += "\n";
}

cols = "";
String temp = "";
//then the north-south weights
for (int row = 1; row < col_weights.length; row++) {
        for (int col = 0; col < col_weights[row-1].length; col++) {

         try {
                 sWeight = JOptionPane.showInputDialog(null,
                                "Enter the weight b/w row " + (row-1)
                                + " and col " + col + " and row " + row +
                                " col " + col, "North-South Weights",

                                JOptionPane.INFORMATION_MESSAGE);
                } catch (HeadlessException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        return;
                }

                 try {
                        col_weights[row-1][col] = Integer.parseInt(sWeight);
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                        return;
                }
```

```
                    cols += sWeight + "  ";
    }
            cols += "\n";
  }

  txtArea1.setText(rows);
  txtArea2.setText(cols);
  dpg.setMatrix(matrix);
  dpg.setRow_weights(row_weights);
  dpg.setCol_weights(col_weights);

  initializeScores();

}



public static void initializeScores(){
        //start by setting the first cell score to 0
        matrix[0][0].setScore(0);
        int score = 0;

        //set the scores for each cell
        //at first column ie column 0 and row row
        for(int row = 1; row < nRows; row++){
                score = matrix[row-1][0].getScore() + col_weights[row-1][0];
                matrix[row][0].setScore(score);
        }

        //then set the scores for each cell
        //at first row ie column col and row 0
        for(int col = 1; col < nCols; col++){
                score = matrix[0][col-1].getScore() + row_weights[0][col-1];
                matrix[0][col].setScore(score);
        }

}

public static void fillMatrix() {
        int aboveWt, leftWt = 0;
  for (int row = 1; row < matrix.length; row++) {
    for (int col = 1; col < matrix[row].length; col++) {
      Cell currentCell = matrix[row][col];
      Cell cellAbove = matrix[row - 1][col];
```

```
        Cell cellToLeft = matrix[row][col - 1];

        aboveWt = col_weights[row-1][col];
        leftWt = row_weights[row][col-1];

        fillCell(currentCell, cellAbove, cellToLeft,
                    aboveWt, leftWt);
      }
    }
}


public static void fillCell(Cell currentCell, Cell cellAbove,
                    Cell cellToLeft, int aboveWt, int leftWt) {
              int aboveScore = cellAbove.getScore() + aboveWt;
              int leftScore = cellToLeft.getScore() + leftWt;
              Map<String, Cell> prevCellMap =
                                              new HashMap<String, Cell>();
              int cellScore = 0;
              Cell prevCell;
              if (leftScore >= aboveScore) {
                if (leftScore > aboveScore) {
                  cellScore = leftScore;
                  prevCell = cellToLeft;
                  prevCellMap.put(Constant.CELL_TO_LEFT, cellToLeft);
                } else {
                       //ideally if they are the same the two
                       //should both be set as pointers
                 //leftScore == aboveScore
                  cellScore = leftScore;
                  prevCell = cellToLeft;
                  prevCellMap.put(Constant.CELL_TO_LEFT, cellToLeft);
                  prevCellMap.put(Constant.CELL_ABOVE, cellAbove);
                }
              } else {//aboveScore > leftScore

                  cellScore = aboveScore;
                  prevCell = cellAbove;
                  prevCellMap.put(Constant.CELL_ABOVE, cellAbove);
              }
              currentCell.setScore(cellScore);
              currentCell.setPrevCell(prevCell);
              currentCell.setPrevCellMap(prevCellMap);
```

```
}


public static String doTraceback() {
        StringBuffer buf = new StringBuffer();
        Cell currentCell = matrix[nRows - 1][nCols - 1];
        Cell prevCell;
        int row = 0;
        int col = 0;
        score = currentCell.getScore();
        while(currentCell != null &&
                        currentCell.getScore() > 0)
        {
                prevCell = currentCell.getPrevCell();
                row = currentCell.getRow();
                col = currentCell.getCol();
                buf.insert(0, "[" + row + "," + col + "]");
                currentCell = prevCell;

        }

        buf.insert(0, "[0,0]");
        return buf.toString();
}

public void doDPWeights(){
        matrix = new Cell[nRows][nCols];
        row_weights = new int[nRows][nCols-1];
        col_weights = new int[nRows][nCols];

        //now compute the other weights
        initialize();
        fillMatrix();
        String wtsStr = doTraceback();
        String results = "The longest path through the matrix as " +
        "obtained by Dynamic Programming is...\n" + wtsStr
                                + "\n" +
                                "The total length of the path is " + score;
        String ruler = "\n===========================================\n";
        txtArea3.append(ruler + results);
}
```

The Graphical output

The graphical output was also created within the same doDPWeights() function that created the functionality of the non-graphical output. The graphical functionality is encapsulated within the DPGraphics class which serves as both a panel and event listener that handles the high-level event listening and actions and contains the main drawing panel. The DPGraphics class has an overloaded constructor to handle either the LCS module or the Weights module.

It receives the user interaction from the container app deals with those action or mouse events. This DPGraphics object that handles Weights functionality delegates the heavy lifting drawing to another panel, the main drawing panel called WeightsDrawPanel. It is this class that contains all the drawing capabilities that draw the Weights matrix. It does most of its drawing in a method called drawMatrix() which takes as input parameters, array of arrays of Weights Cell objects.

Drawing the Weights Matrix

Drawing the grid of the Weights matrix occurs very similar to the way LCS is drawn. The vertices are drawn first by drawing the cells of the matrix. Then the west-east edges are drawn followed by the north-south weights. Then the weights are drawn as numbers beside and above the edges and numbers in the vertices as solutions are computed. The arrows are drawn as the solutions are produced and longest paths to the sink are found.

```
public void drawMatrix(Cell[][] matrix, int row_weights[][],
                int col_weights[][]){

        if (matrix == null ||
                row_weights == null || col_weights == null)
                return;
        int x = 30;
        int y = 30;
        int x1 = 0;
        int x2 = 0;
        int y1 = 0;
        int y2 = 0;
        int x3 = 0;
        int y3 = 0;
        int width = 50;
        int height = 50;
        int space = 100;
        int row;
```

```
int col;
String score;
Stroke prevStk;
Stroke boldStk = new BasicStroke(2f);
String wts = "";
Map<String, Cell> prevCellMap = null;
area = new Dimension();
g2d = (Graphics2D)gr;
Cell[][] copyMatrix = matrix;
row = copyMatrix.length;
col = copyMatrix[0].length;
int lastScore = copyMatrix[row-1][col-1].getScore();
String sScr = String.valueOf(lastScore);
          int dim = (sScr.length() * 2) + g2d.getFontMetrics().getHeight());
width += dim;
height += dim;
space += dim;
for (row = 0; row < copyMatrix.length; row++) {
    for (col = 0; col < copyMatrix[row].length; col++) {
      Cell currentCell = copyMatrix[row][col];
      Cell prevCell = currentCell.getPrevCell();
      prevCellMap = currentCell.getPrevCellMap();

      if(row == 0 && col > 0)
          prevCellMap = null;
      if(col == 0 && row > 0)
          prevCellMap = null;
      prevStk = g2d.getStroke();
         //if the prev cell is null just draw the current cell
      if (prevCellMap == null  || prevCellMap.isEmpty())
      {
          g2d.drawOval(x, y, width, height);

          if(col < matrix[row].length - 1 && row == 0){
             x1 = x + width;
             y1 = y + (height/2);
             x2 = x + space ;
             y2 = y1;
             g2d.setStroke(boldStk);
             g2d.drawLine(x1, y1, x2, y2);
                    //draw the west-east weights
                    wts = String.valueOf( row_weights[row][col] );
                  x3 = (( x1 + x2 ) / 2 ) - (wts.length()*2);
                      y3 = y1 - 10;
```

```
            g2d.drawString(wts, x3, y3);
            //now draw the arrows
                x1 = x2 - 5;
                y1 = y2 - 5;
                x3 = x2 - 5;
                y3 = y2 + 5;
                g2d.drawLine(x1, y1, x2, y2);
                g2d.drawLine(x2, y2, x3, y3);
            g2d.setStroke(prevStk);
}

//if the row is not 0 draw the north-south line
if (row > 0 && col == 0){
    //draw the north-south line
    x1 = x + (width/2);
          y1 = y - space + height;
          x2 = x1;
          y2 = y1 + space - height;
          g2d.setStroke(boldStk);
          g2d.drawLine(x1, y1, x2, y2);

          //draw the north-south weights
          //g2d.setStroke(prevStk);
          wts = String.valueOf( col_weights[row-1][col] );
    x3 = x1 + 10;
    y3 = ( y1 + y2 ) / 2;
    g2d.drawString(wts, x3, y3);

          //draw the north-south arrow head
          x1 = x2 - 5;
          y1 = y2 - 5;
          x3 = x2 + 5;
          y3 = y2 - 5;
          g2d.drawLine(x1, y1, x2, y2);
          g2d.drawLine(x2, y2, x3, y3);
          g2d.setStroke(prevStk);
}

if(col < matrix[row].length - 1 && row != 0){
x1 = x + width;
    y1 = y + (height/2);
    x2 = x + space ;
    y2 = y1;
```

```
                g2d.drawLine(x1, y1, x2, y2);
            //draw the west-east weights
                    wts = String.valueOf( row_weights[row][col] );
                    x3 = (( x1 + x2 ) / 2 ) - (wts.length()*2);
                    y3 = y1 - 10;
                g2d.drawString(wts, x3, y3);
                    g2d.setStroke(prevStk);
    }

}
    else //draw the currentcell and get the direction of the
       //prevcell so we can draw arrows pointing to it
       //and then draw the score and the weights
    {
            g2d.drawOval(x, y, width, height);
            if(col < matrix[row].length - 1 ){

            //now draw the west-east line
            x1 = x + width;
                    y1 = y + (height/2);
                    x2 = x + space ;
                    y2 = y1;
                    g2d.drawLine(x1, y1, x2, y2);
                    //draw the west-east score
                    wts =
                    String.valueOf( row_weights[row][col] );
                    x3 = (( x1 + x2 ) / 2 ) - (wts.length()*2);
                    //x3 = x1 - wts.length();
                    y3 = y1 - 10;
                    g2d.drawString(wts, x3, y3);
            }
            //if the row is not 0 draw the north-south line
            if (row > 0){
                     //draw the north-south line
                            x1 = x + (width/2);
                            y1 = y - space + height;
                            x2 = x1;
                            y2 = y1 + space - height;
                                  g2d.drawLine(x1, y1, x2, y2);
                     //draw the north-south weights
                     wts =
                              String.valueOf( col_weights[row-1][col] );
                     x3 = x1 + 10;
                     y3 = ( y1 + y2 ) / 2;
```

```
                g2d.drawString(wts, x3, y3);
            }
        //if the prev cell is above draw arrow again with
        //bold stroke

                if(prevCellMap.get(Constant.CELL_ABOVE) != null) {
                  x1 = x + (width/2);
                  y1 = y - space + height;
                  x2 = x1;
                  y2 = y1 + space - height;
                  g2d.setStroke(boldStk);
                  g2d.drawLine(x1, y1, x2, y2);
                  //draw the north-south arrow head
                  x1 = x2 - 5;
                  y1 = y2 - 5;
                  x3 = x2 + 5;
                  y3 = y2 - 5;
                  g2d.drawLine(x1, y1, x2, y2);
                  g2d.drawLine(x2, y2, x3, y3);
                  g2d.setStroke(prevStk);
                 }
          if(prevCellMap.get(Constant.CELL_TO_LEFT) != null)
          {
                  x1 = x;
                  y1 = y + (height/2);
                  x2 = x - space + height;
                  y2 = y1;

                  g2d.setStroke(boldStk);
             g2d.drawLine(x1, y1, x2, y2);
                  //now draw the arrow heads
                   x2 = x1 - 5;
                   y2 = y1 - 5;
                   x3 = x1 - 5;
                   y3 = y1 + 5;
                   g2d.drawLine(x1, y1, x2, y2);
                   g2d.drawLine(x1, y1, x3, y3);
                   g2d.setStroke(prevStk);
        }
    }

    //draw the score
    score = String.valueOf(currentCell.getScore());
    x1 = x + (width/2) - (score.length()*2);
```

```
                y1 = y + (height/2);
                g2d.drawString(score, x1, y1);
                x += space;

              }
              if (row == 0)
              area.width += x;
              x = 30;
              y += space;

             }
 area.height = y;
}
```

DISCUSSION

DPMax for Weights set out to implement an algorithm that solves the Manhattan Tourist Problem and more. It would also provide as many of the longest paths existing in the grid. If indeed there are 2 or more longest paths, it means the tourist can have the luxury of choosing only the very best attractions given the situation.

DPMax for Weights not only makes this task simple but also displays the one or more longest paths on a grid which can be examined visually or even printed out for future use and reference.

DPMax for Weights can be upgraded and modified to find important use in the following ways:

1. Transportation

The field of transportation can benefit greatly from the Weights algorithm. The algorithm can be used to find the longest paths between 2 points and use that calculation to create schedules for traveling, and for calculating costs for accounting and finance. This use is highlighted in the topic covered in this paper.

2. Travel and Tourism

The algorithm can also be used to create schedulers for the tourism industry. Travelers can use this schedule to build their personal tourism schedules in any region they visit. Airlines can compute the distances and time to travel those distances. This information can then be used to calculate the amount of fuel and other costs needed for travel. This use is also highlighted in the topic covered in this paper.

3. Construction and Civil Engineering

The Weights algorithm can be used in construction of infrastructure like bridges, passes and so on. There may be a need to find the paths in the building of these structures.

4. Economics, Business and Finance

Dynamic Programming by itself is an invaluable technique for numerous economics and finance procedures such as cost accounting, budgeting and profit and loss accounting. Furthermore, there is immense use of DP algorithms for computing largest and smallest costs (longest and shortest paths) in the daily, monthly and annual transactions of business ventures (Bradley et. al.).

5. Others

The algorithm can also be used in various industries to calculate correct spacing in manufacturing, construction, building, and so on.

The Shortest Paths Algorithm

This paper has looked at the problem of finding the longest paths between a start and finish points. But there is also the need to find the opposite: the shortest paths between 2 points on a grid. I shall call this problem the *Impatient Tourist Problem*, ITP. This problem is an implementation of the famous *Shortest Paths Problem*. It is about a tourist who is in a hurry to get from point A to point B but wishes to get through them in the shortest amount of time, thereby seeing the best possible attractions, but in the shortest time, and path, possible (Bradley et. al., Chapter 11).

The solution to this problem is quite simple, considering we have gone through the algorithm for the longest paths problem. It must be noted that the ITP is by no means the same thing as the Traveling Salesman Problem, which goes through all cities without repetition and returns to the starting city (Reingold et. al.). ITP is simply the Manhattan Tourist, but instead of doing the maximum between two paths, it seeks the minimum between the 2 possible paths. The ITP has the same number of subproblems as the MTP, given the same number of rows and columns. This means that it will have the same number and weights of its edges, west-east and north-south, as the MTP. It will build the very same grid structure as the longest path problem it solved. And it will create the same number of vertices as the longest path problem, LPP. It solves the same grid, but it solves the decision variables at each stage of solving a subproblem slightly differently. It performs a minimum operation at each stage, rather than a maximum one. That is, it computes the value of the vertex, and path, that is smaller, rather than bigger, unlike the LPP.

So, the algorithm to find the shortest paths between source and vertex is an expected modification of the MTP. The difference is shown on line (6).

function ImpatientTourist()

To Find: The shortest paths between 2 points on a grid (Graph) of intersecting edges and vertices.

H[] – A 2-Dimensional array of weights of edges that run from west to east (horizontal)

V[] – A 2-Dimensional array of weights of edges that run from north to south (vertical)

n = length of the array *H[]*

m = length of the array *V*[]

(1) *G*[0, 0] = 0

(2) for i = 0 to n

G[i, 0] = G[i -1, 0] + *V*[i, 0]

(3) for j = 0 to n

$G[0, j] = G[j - 1, 0] + H[0, j]$

(4) for i = 1 to n

(5) for j = 1 to m

(6) $\mathbf{L_{i, j} = \min (L_{i,-1, j} + V[i, j], L_{i, j-1} + H[i, j])}$

(7) return $L_{n, m}$

IMPROVEMENTS

The improvements to the DPMax for Weights program are the same as its LCS counterpart, which have been discussed in the LCS paper.

CITATIONS

Baase, Sara, and Allen Van Gelder. Computer Algorithms: Introduction to Design and Analysis. 3rd ed., Addison-Wesley Longman, 2000.

Baxevanis, Andreas D., and B. F. Francis Ouellette, editors. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. 3rd ed, Wiley, 2005.

Bellman, Richard, and Stuart Dreyfus. *Dynamic Programming*. 1. Princeton Landmarks in Mathematics ed., With a new introduction, Princeton University Press, 2010.

Bradley, Stephen P., et al. Applied Mathematical Programming. Addison-Wesley Pub. Co, 1977.

Chong, Jike, et al. *Dynamic Programming Pattern*. https://patterns.eecs.berkeley.edu/?page_id=416. Accessed Apr. 2009.

Cooper, Leon, and Mary W. Cooper. *Introduction to Dynamic Programming*. 1st ed, Pergamon Press, 1981.

Cormen, Thomas H., editor. *Introduction to Algorithms*. 3rd ed, MIT Press, 2009.

Dasgupta, Sanjoy, et. al. *Algorithms*. McGraw-Hill Higher Education, 2008.

Denardo, Eric V. *Dynamic Programming: Models and Applications*. Dover Publications, 2003.

GeeksforGeeks. *Overlapping Subproblems Property in Dynamic Programming | DP-1* https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/. Accessed November 13, 2018.

Mount, David W. *Bioinformatics: Sequence and Genome Analysis*. 2nd ed, Cold Spring Harbor Laboratory Press, 2004.

Needleman, S. B., and C. D. Wunsch. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins.” *Journal of Molecular Biology*, vol. 48, no. 3, Mar. 1970, pp. 443–53.

Reingold, Edward M., et al. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

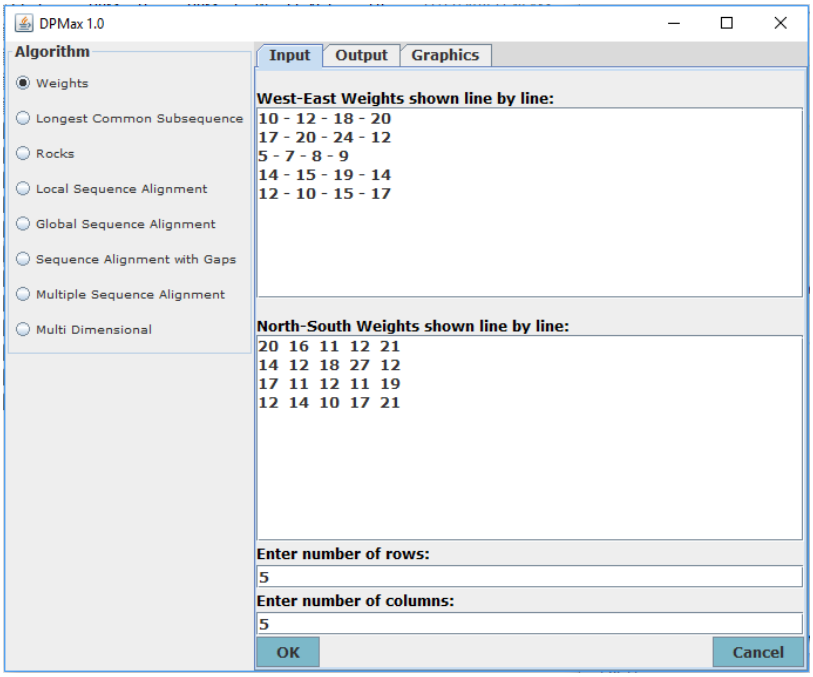
Smith, T. F., and M. S. Waterman. “Identification of Common Molecular Subsequences.” *Journal of Molecular Biology*, vol. 147, no. 1, Mar. 1981, pp. 195–97.

Skiena, Steven S. *The Algorithm Design Manual*. 2nd ed, Springer, 2010.

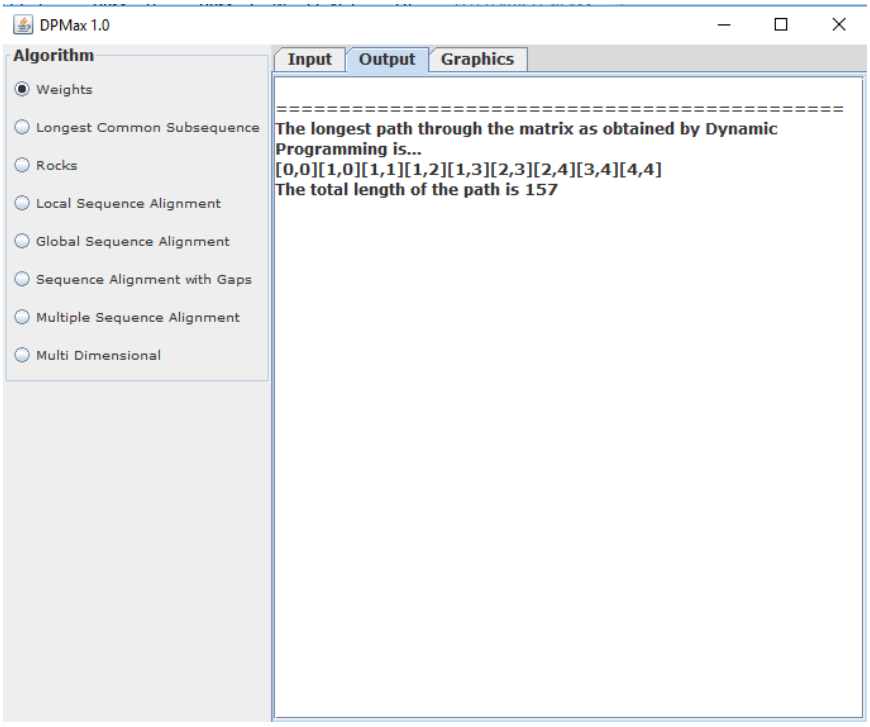
Appendix A

Longest Path (Manhattan Tourist) Problem Through a 5x5 grid matrix

1. Input tab



2. Output tab



3. Graphics tab